LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-212323

Scaling Up Data-Centric Middleware on a Cluster Computer

D. T. Liu, M. J. Franklin, J. Garlick, G. M. Abdulla

May 16, 2005

Supercomputing 05
Seattle, WA, United States
November 12, 2005 through November 18, 2005

Scaling Up Data-Centric Middleware on a Cluster Computer

David T. Liu, Michael J. Franklin
UC Berkeley, CS Division
Berkeley, CA 94708
{dtliu,franklin}@cs.berkeley.edu

Jim Garlick, Ghaleb M. Abdulla
Lawrence Livermore National Laboratories
Livermore, CA 94551
{garlick1,abdulla1}@llnl.gov

ABSTRACT

Data-centric workflow middleware systems are workflow systems that treat data as first class objects alongside programs. These systems improve the usability, responsiveness and efficiency of workflow execution over cluster (and grid) computers. In this work, we explore the scalability of one such system, GridDB, on cluster computers. We measure the performance and scalability of GridDB in executing data-intensive image processing workflows from the SuperMACHO astrophysics survey on a large cluster computer. Our first experimental study concerns the scale-up of GridDB. We make a rather surprising finding, that while the middleware system issues many queries and transactions to a DBMS, file system operations present the first-tier bottleneck. We circumvent this bottleneck and increase the scalability of GridDB by more than 2-fold on our image processing application (up to 128 nodes). In a second study, we demonstrate the sensitivity of GridDB performance (and therefore application performance) to characteristics of the workflows being executed. To manage these sensitivities, we provide guidelines for trading off the costs and benefits of GridDB at a fine-grain.

1. INTRODUCTION

In recent years, scientists in many disciplines, including physics, astronomy, and biology, have looked to grid computing to accommodate their ever-increasing appetite for digital data generation, transformation and analysis [12, 16].

In response to the demand for better grid computing tools, computer scientists have proposed a number of "data centric" workflow middleware systems [15, 20, 21, 9, 7]. These systems treat data as first class objects alongside workflow programs and exploit their understanding of data to improve improve user-productivity. Often, these systems are equipped with simple interfaces or languages that improve the usability, programmability, and responsiveness of grid resources. Some also seek to automate otherwise tedious, but essential, tasks, such as data provenance. In this paper, we study the performance and scalability of our system, GridDB, from UC Berkeley [15].

Figure 1 depicts the role of GridDB. A scientist interacts with grid infrastructure (in this case, a cluster computer) through GridDB, rather than directly, to secure higher-level interfaces and benefits. These benefits include:

- Simple interfaces for defining and executing computations.

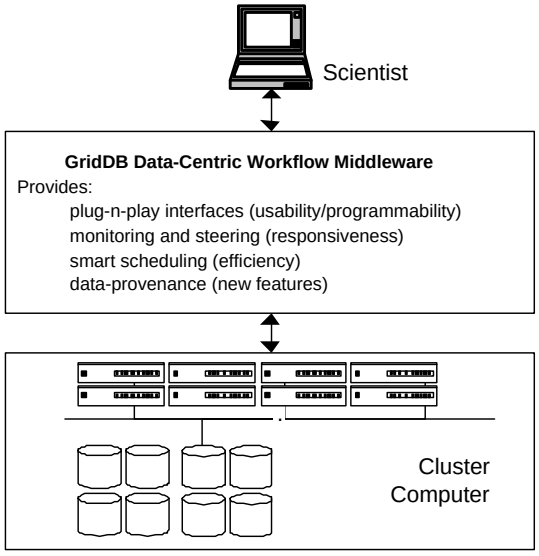

Figure 1: This paper focuses on the performance and scalability of GridDB on a cluster computer.

- Monitoring and steering of long-running computations.
- "Smart" scheduling to automatically maximize the utility of available resources.
- Data provenance tracking and querying, allowing users to determine the origins of a computation.

While there has been workflow middleware research focused on distributed computation, where data placement and fast file transfer is of paramount importance [13, 19], our focus in this paper is on how GridDB may be scalably implemented on the cluster platform.

An important difference between cluster computers and "grid computers" is that data from one computation node can be accessed by other computation nodes through a network file system, obviating the need for file transfers. As a result, clusters can support data-intensive (read and write many bytes), as well as purely compute-intensive (consume many compute cycles) workflows [14]. Additionally, users no longer have the incentive to optimize file-transfer precisely, its human cost may not justify its performance benefit. Finally, with the file transfer bottleneck removed, workflow middleware may emerge as the next bottleneck.

In this work, we retrofit our previous implementation of GridDB [15], specializing it towards the cluster environment.

Using our new implementation, we perform experimental studies with data-intensive astrophysics image processing pipelines from the SuperMACHO Project [6]. Using MCR, a large cluster computer at LLNL, we make the following contributions:

- We show that, even in the absence of file copies, and in the presence of a transaction-processing database system, file system operations represent the first tier bottleneck in GridDB. We circumvent this bottleneck with a pair of optimizations, more-than-doubling the scalability of our system. The optimized implementation easily handles a 128-node cluster.

- We show that the scalability of a system such as GridDB will be highly sensitive to the characteristics of the workflows it is executing. We define a property of workflow programs, the *middleware-load*, to capture the amount of strain a workflow program places on middleware. We provide guidelines for minimizing middleware-load, while maximizing the benefits gained from middleware.

- Finally, we demonstrate that programs can exhibit different middleware-load on two popular global file-systems, NFS and Lustre. We characterize GridDB's performance and scalability in both cases.

2. BACKGROUND

We start our discussion by providing background information. This section summarizes information from a previous work [15] that is relevant to our performance study. This section describes the workflows that GridDB supports, the benefits it provides, and how information about a workflow is conveyed from a user to GridDB (modeling).

2.1 Workflows

In this section, we describe GridDB's workflow model. Workflows are sets of modular *programs*, interacting to transform an available input into a desired output. Each program converts input data to output data. Input data comes from one or more[1] *filesets*, or sets of files, and a set of command-line *parameters*. Output consists of one or more filesets. A program G *depends* on another program F if F creates a fileset used by G. Within such a relationship, we refer to F as the *parent* and G as the *child*. A process is an instantiation of a program, or a combination of a program and an instance of its inputs.

Figure 2 shows an example workflow from the SuperMACHO image processing pipeline. We use this pipeline in the majority of our experimental studies. The overall goal of the pipeline is to extract a set of sky objects from a telescope image. The pipeline proceeds as a two-step process. First, the mscpipe program transforms and splits a Telescope Image fileset into 16 reduced image filesets ($\{RedImg_1, \ldots, RedImg_{16}\}$. Each of these reduced images can be processed in parallel by the photpipe program to extract sky objects. Both programs, mscpipe and photpipe, use auxiliary fileset inputs (Reference Images and Calibration Images, respectively). These image processing programs are written in a combination of C and perl. In our test environment (described in Section 4), the average execution times of the two programs mscpipe and photpipe are 273 and 342 seconds, respectively.

[1] filesets may come from more than one source

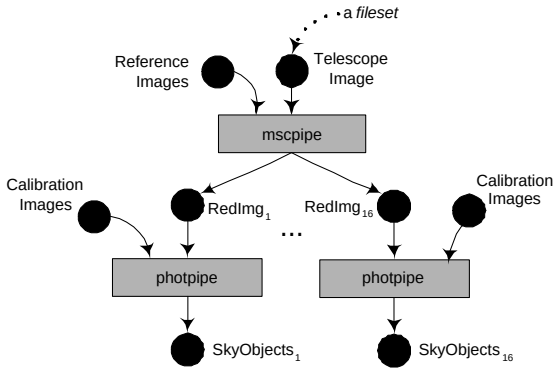

Figure 2: FullPipe: The SuperMacho image processing workflow

2.2 Benefits of GridDB

By executing workflows through GridDB, a scientist gains a host of benefits that improve his productivity:

- **Automatic Execution**: Processes are automatically prepared and launched. Dependencies between processes are automatically enforced.

- **Data Provenance**: Given a selected data product, a user should be able to examine the programs and data that led to its existence. For example, the provenance of $SkyObjects_{16}$ in Figure 2 includes the filesets RedImg_{16}, Calibration Images, Reference Images, and Telescope Image and the programs mscpipe and photpipe. o

- **Efficient Recomputation (or Computation Caching)**: Commonly, scientist needs to rerun a computation with a parameter change. In many circumstances, re-generation of data products can be achieved without complete workflow re-execution. Armed with workflow information, GridDB can identify these opportunities. For example, suppose that a user would like to rerun the computation of Figure 2 with a different set of Calibration Images. In this circumstance, the process `mscpipe` may avert re-execution, as its results would be unaffected by a change in Calibration Images. Generally speaking, this is particular advantageous when the process whose execution was eliminated is resource-intensive.

- **Scheduling**: Studies have shown that intelligent resource allocation can make significant differences in execution efficiency. GridDB, being aware of resource availability, workflow execution requirements, and data product desirability, can make intelligent scheduling decisions.

- **Computational Monitoring and Steering**: workflow processes submitted for cluster execution are often long-running. Often, it is useful to monitor workflow execution and steer it by prioritizing sub-workflows that produce the most interesting results. For example, LSST, an astronomy project that we collaborate with (the LSST [5]) needs to execute a set of verification processes upon spotting a suspected supernova. In such cases, GridDB can automatically divert cluster resources towards these verification processes

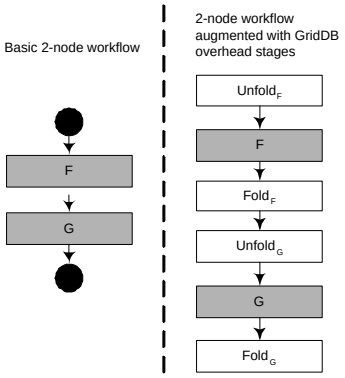

Figure 3: A basic 2-node workflow and an augmented workflow with the unfold and fold overhead stages.

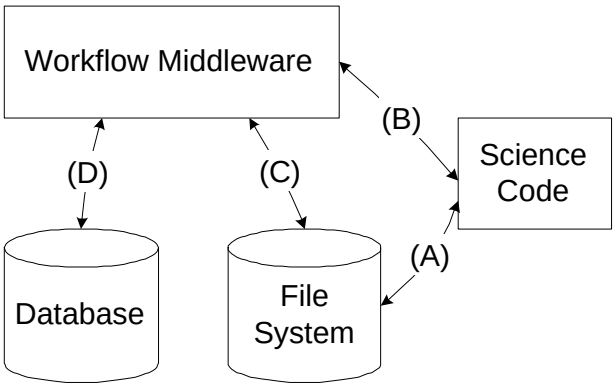

Figure 4: The Software Component Ecosystem

2.3 GridDB Modeling

To obtain the benefits mentioned in the previous section, GridDB must be given information about the workflows it is to execute. In GridDB, the description of workflows is decoupled from their execution. The first is performed by the *modeler* role and the second by the *analyst*. The modeler performs setup by defining workflows and their interfaces, include the names that types of filesets that are produced and consumed by each program. After definition of these workflows, the analyst may pull these workflows "off-the-shelf," applying them to collections of inputs. Workflow evaluation automatically proceeds by dispatching processes to cluster (or grid) computing nodes. Through the modeler's efforts, the analyst reaps the benefits mentioned above.

3. GRIDDB SYSTEM OVERVIEW

In Section 2.2, we described a suite of benefits to increase a scientist's productivity when executing workflows over a cluster. In this section, we describe, at a conceptual level, how these features are provided by GridDB. Ultimately, GridDB carries out a set of *tasks*, which occur in stages before and after the execution of each processes. These "overhead" stages, which we call *unfold* (before execution) and *fold* (after execution) involve middleware execution, interleaved with database queries and transactions and file system linking operations, as depicted in Figure 3.

The relationships between GridDB, database, file system and science codes is shown in the component ecosystem of Figure 4, which we proceed to described. The science code, reads and writes persistent data from and to the file system (arc A). GridDB prestages data into a science code process' working directory (arc C) and dispatches it for execution onto a cluster node (arc B). Tables that catalog monitoring information, provenance information, and tuple data are inserted and retrieved by GridDB (arc D), and stored in the database.

In the next part of this section, we describe how this scheme emerges. Provisioning of the target features can be factored into a set of responsibilities. These responsibilities can be implemented as file system calls and database queries/transactions that are carried out either before (in the unfold stage) or after (in the fold stage) process execution.

We trace the cause-effect relationships that lead from high-

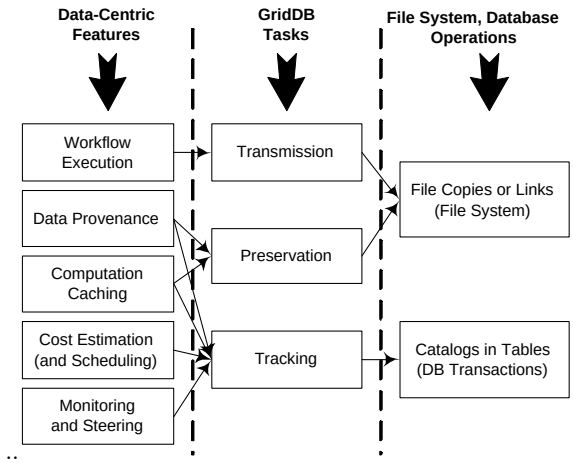

Figure 5: The mapping from responsibilities to tasks, to operations. Operations are carried out in GridDB overhead stages.

level, user-centric features down to low-level software operations. Ultimately, the features we would like to provide result in file system operations and database transactions that occur before and after each program execution.

3.1 Features, Task and Operations

Figure 5 summarizes the relationships between features, tasks and operations. We describe each of the three tasks, transmission, preservation and tracking, below. In each case, we explain the tasks, the features they support, and the file system or database operations used to implement them.

To provide workflow execution capabilities, the middleware must perform data *transmission*, or the transfer of filesets from one process' working space to another process' working space. These can either be carried out as copies or links from one process' working directory to the next.

Two other tasks, *preservation* and *tracking*, jointly provide the other services, data provenance, computation caching, cost estimation (which is needed for intelligent scheduling), monitoring and steering.

Preservation is an enforcement function — when a process, P, uses a fileset, it must *not* modify the fileset. Other-

wise, the data provenance of P is lost, and the consumed (and modified) data product cannot be deterministically reused (i.e., computation caching) by other processes that require the same fileset. Preservation can be provided in one of two ways, depending on how transmission is carried out. If transmission is carried out with file copies, preservation is automatically provided, even if the copy is modified, the original is preserved. If transmission is provided through file linking, GridDB must also set the files in the fileset to be read-only as modifications through a link will taint the original fileset. In Section 4.1, we discuss the advantages and disadvantages of alternative methods for file transmission.

Tracking is implemented through a collection of mappings: from filesets to their parent processes, from processes to the inputs used to create them, and from processes to resources expended in executing the processes. These mappings are best stored and queried in table structures. We implemented this functionality using a relational database for two reasons: first, it helped us prototype quickly and second, these catalogs may become large, and must remain persistent. Both requirements are aptly handled by off-the-shelf databases.

Because of these these interactions between middleware, file system, and database, the time in an overhead stage can be partitioned between GridDB execution, and calls to the file system and database.

3.2 Programs and Middleware-Load

As we demonstrate experimentally in Section 6.4, GridDB's performance and scalability is sensitive to the load placed upon it by the programs it is running. Therefore, we characterize each program with a *middleware-load*, or amount of strain that the program places upon GridDB.

As described in Section 2.3, a modeler includes with each program specification a list of data tuples and filesets that belong to the input of each program. These specifications are used to stage inputs for processes using the program, as well as collect and catalog the process' outputs. Since the inputs of programs need to be retrieved and staged by GridDB, and the outputs need to be cataloged and stored, the programs middleware-load is directly proportional to both. An additional factor to *middleware-load* is the runtime of the program. Short-running programs require middleware action more frequently than long-running programs, and therefore present a higher load to the middleware. Pulling these factors together, we can describe a program P's middleware-load(ML) on a system as:

$$ML_P = \frac{cost(numQueries_P) + cost(numLinks_P)}{runtime_P}$$

A program with a higher ML_P will more difficult to handle, and consequently, GridDB's performance and scalability will be inversely proportion to the ML_P of the workflows it shepherds. In Section 6.4, we show several examples of programs and their middleware-loads.

4. THE CLUSTER ENVIRONMENT

In this paper, we focus our efforts on cluster computers. A cluster computer is a large computer created from a collection of small, often commoditized, components (cpus, disks, memory, network interconnects). Commoditization has made cluster computers the platform of choice for cost-effective scientific computing [18].

An important characteristic of cluster computing that we exploit is that file transmission can be achieved with a link, rather than a copy, as is required in distributed grid computing. There are two key implications to this: first, it enables a broad class of applications, which are not only compute-intensive, but also data-intensive (such as the SuperMACHO image processing application that we work with) [14]. Second, it qualitatively enables "lightweight modeling" in GridDB, easing the burden on the modeler.

4.1 Links Instead of Copies

In section 3.1, we described file transmission as one of the responsibilities that GridDB was responsible for. As previously mentioned, file transmission on a cluster can occur through file-linking, rather than file copying. With the added caveat that the linked-to file must be made read-only. This optimization has a tremendous benefit — the performance of file transmission is no longer dependent on the size of the file.

The naive instantiation of transmission through linking is the "leaf-linking" transmission method of Figure 6. Leaf-linking replicates the directory structure of filesets transferred from parent to child, but links files instead of copying them. We initially tried the 'root-linking' method to optimize further, but ran into a problem: the receiving program was not allowed to add files to any directories used by the input filesets. Adding files would violate the preservation of the input filesets, as illustrated by the addition of file 4 to directory B in Figure 6. Therefore, we had to resort to leaf-linking, which performs worse than root-linking, as we demonstrate in Section 6.2.3.

4.2 Lightweight Modeling

The cluster environment also provides users with the option of reducing their modeling effort by specifying filesets at a coarse-grain. Modelers, in specifying which files are created, may specify them at a coarse grain. For example, suppose that a parent program, F, creates two files with paths `out/1` and `out/2`, only one of which is used by a child program G. The modeler can simply specify that `out` is exported from F to G, without understanding exactly which files are being used. Such a coarse specification of program interfaces requires less understanding of programs, which is potentially expensive, and is easier to maintain. As an example of ease-of-maintenance, suppose F later creates a new file `out/3` that is needed by G; the modeler's prior specification remains unchanged.

In a distributed scenario, such "sloppy" modeling may be ill-advised. File-transfer is the bottleneck, so transporting files gratuitously may severely impact application performance. On a cluster, file transfers are obviated, so we can refocus our efforts on the human bottleneck.

5. A BASELINE IMPLEMENTATION

GridDB's programming logic is decomposed into a set of *stages* connected by queues (similar in spirit to an event-driven programming model). Stages receive requests, perform some operations, and possibly enqueue requests in into output queues. Each stage can be served by multiple threads. This is used to take advantage of concurrency when a stage is long-running.

An architectural diagram of the stages of GridDB is shown in Figure 7. There are four stages, each connected by queues. We describe the stages by walking through the life of a request. A client submits a request for a set of computations to the *Req* stage. The request consists of a workflow, and a set of inputs to the workflow. It is decomposed into a set of processes. Processes are sent to the $Unfold$ stage, wherein

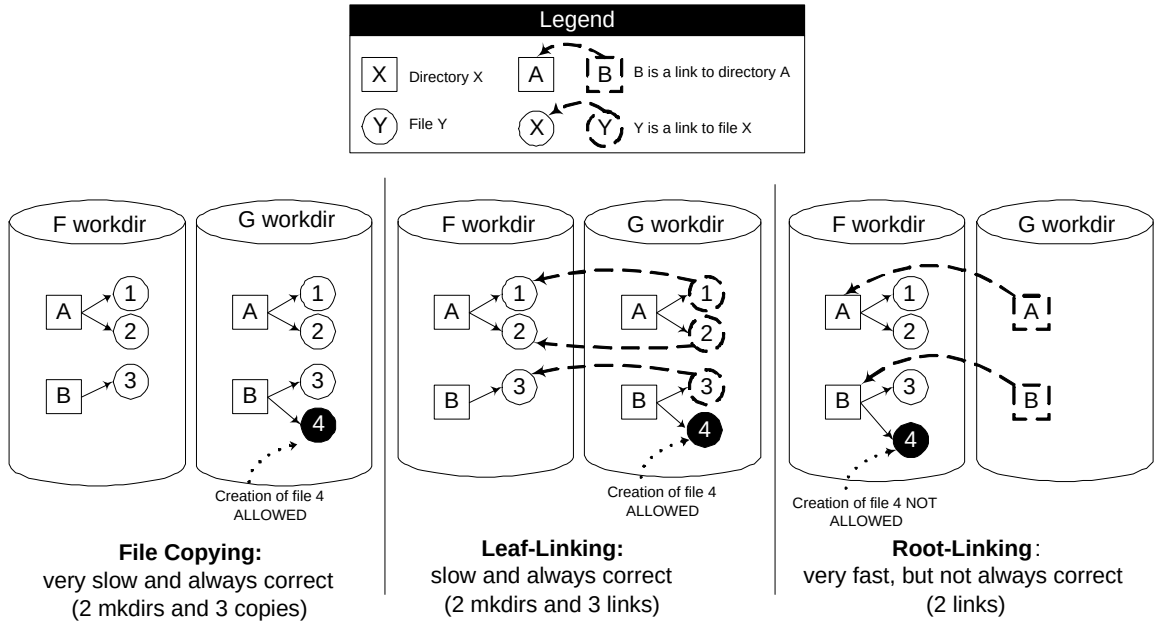

Figure 6: File Transmission Methods between parent process F and child process G. Copying provides transmission and preservation, but is unnecessarily expensively on a cluster. Leaf-linking replicates the directories of filesets, but links files. Root-linking automatically links to target directories but does not allow creation of files in the directories, as may be required by child process G (file 4 cannot be created using root-linking).

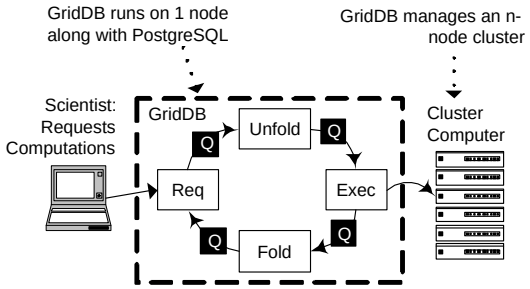

Figure 7: Stages in GridDB Processing.

a working directory for the process is created on the file system. When unfolding is completed, the process is sent to the *Exec* stage, where the process is assigned to a node. The node executes a computation over the process' working directory, modifying it and creating output filesets. When the process completes, the *Exec* stage enqueues a request for the *Fold* stage to post-process the request. This includes all of the following activities: collecting output FileSets, extricating children processes that require the current process' output, logging information tracking information into the database.

Each process goes through its three phases, unfold, exec and post. The exec stage is spent on a cluster node, while the unfold and post stages are spent in GridDB. Bottlenecks may occur in a couple of ways. If a job spends too much in its unfold phase, it is not allowed to proceed to its exec stage, where it occupies a cluster node. Alternately, if the unfold stage of a parent process requires too much time, a child process may be prevented from executing, as the child blocks on the parent's input. This is how GridDB scalability may be an issue: GridDB must work fast enough to keep processes moving through the unfold and fold stages, in order to keep the n-node cluster utilized. If the unfold and fold stages require too much time, the middleware will fail to occupy the cluster fully.

Our implementation was written in java, consisting of 5059 non-commented source statements. Our database of choice was PostgreSQL. We benchmarked four open-source databases, PostgreSQL [4], MySQL [3], hsqldb [2] and Apache Derby [1]. MySQL was actually the fastest with postgresql in a close second. We chose PostgreSQL, however, for its more complete implementation of SQL (most notably, it included views and subqueries). In the end, the small performance penalty did not matter. The latter two DBMS's offered an opportunity for tight coupling with the GridDB source, as they are written in java. However, their performance was considerably slower than the that of the first two.

One problem that we encountered in using java was the lack of a file linking API call (like `symlink(2)`). This call is used for our linking-based file transmission methods. Because java strives to be platform independent, and the semantics of linking differs across computing platforms, linking API's are not provided. To workaround this, we had to make calls to `ln` to create links. As we will show in our experiments section, Section 6.2, this was a source of GridDB overhead.

6. EXPERIMENTS AND OPTIMIZATIONS

We discuss experimental results across varying implementations, workloads and file system platforms. All experi-

ments were run by isolating an n-node cluster from the MCR cluster computer at LLNL.

We ran three main sets of experiments. In our first set of experiments we discovered that GridDB's baseline implementation has trouble scaling up to 64 nodes. After drill-down, we identified file system linking as GridDB's main bottleneck. After implementing a pair of optimizations, we scaled easily to 128 nodes, more than doubling GridDB's scalability.

In our second set of experiments, we execute a highly intensive workflow to demonstrate that a workflow system like GridDB is sensitive to workflow programs' middleware-load. To gain control over the load imposed by workflow programs upon GridDB, Section 7 provides guidelines for deciding whether or not workflow programs should be decomposed and exposed to GridDB. These decisions are based on the costs and benefits of different modeling alternatives.

In our last set of experiments, we compare GridDB's performance on two separate network file systems, Lustre and NFS. We show that the middleware-load of programs run on NFS is lower than when they are run on Lustre. This is due to two factors: (1) programs run on NFS execute slower than on Lustre and (2) metadata (including linking) operations on Lustre are slower than on NFS. Incidentally, while users will see better overall application performance using Lustre, GridDB will scale better on NFS.

6.1 Experimental Setup

Our experiments were carried out on the MCR cluster computer. MCR is an 11 TFLOPS cluster with 1152 nodes, each with 2 Intel Xeon 2.4 GHz processors and 4 GB of memory. In most of our experiments, we ran our science codes over the Lustre global parallel file system [8]. In our third set of experiments, we run some experiments on the NFS file system.

Our experiments were carried out by allocating blocks of nodes for 6 to 12 hours at-a-time to gain isolation from other processes running on the cluster. To emulate computation on an n-node cluster, we allocated a block of $n + 1$ nodes. GridDB and PostgreSQL were launched on one node (as shown in Figure 7. Afterwards, we submitted a request for processing m images through one of two workflow, which are described in the next section. In all our experiments, we sent twice as many images compared to the number of nodes in the cluster ($m = 2n$). Offered load is always high-enough to keep the cluster occupied. If it is under-utilized, it is because GridDB is becoming a bottleneck. In all of experiments, scheduling was done in "batch-mode." Programs earlier in the pipeline were executed before programs later in the pipeline.

On MCR, we do not have isolation of the network and network file systems. These resources are shared by all processors simultaneously. To minimize contention of network and storage resources, all of our experiments were carried out between 6pm and 6am, when these resources were the least loaded.

6.1.1 Workflows

We used two workflows in our experiments. Our main workflow, the SuperMACHO image processing pipeline (FullPipe), was described in Section 2.1 and is shown in Figure 2. The parameters that affect SuperMACHO's middleware-load is shown in Table 1. Of the two programs in this workflow, mscpipe is more intense. Its unfold phase involves 788 links, a fact that we did not discover until we actually ran the workflow in leaf-linking mode. The fold phase

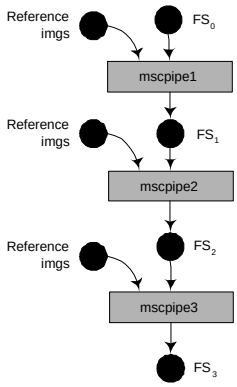

Figure 8: [EXP2] A workflow created from decomposing the mscpipe program of FullPipe into three sub-programs.

of mscpipe involves a large number of database transactions. This is due to the fact that the completion of an mscpipe process involves data structure instantiation (in the database) for the 16 photpipe processes that are spawned by each mscpipe process.

We use a second workflow to demonstrate the sensitivity of GridDB's scalability to workload.

The second workflow, DecomposedPipe, is a workflow created from decomposing the mscpipe program of FullPipe into three parts(mscpipe1,mscpipe2,mscpipe3). It's topology is shown in Figure 8 and its middleware-load characteristics are shown in Table 2. This pipeline, with its shorter program execution times (the average stage length is one-third the average stage length of FullPipe), and comparable linking and query characteristics should be much harder for GridDB to handle. We use it in Section 6.4 to demonstrate that the overhead of middleware (and the performance of applications running on the middleware) is sensitive to a programs middleware-load.

6.2 Exp 1: Tuning GridDB

Our first set of experiments involves the profiling of our baseline implementation. Our baseline scale-up was not good. GridDB became the bottleneck while trying to process 128-images through FullPipe on a 64-node cluster. Our system profiles were surprising — file system metadata operations (rather than database transactions) were responsible for GridDB's laggard performance. We implemented a pair of optimizations to reduce the cost of file-system metadata operations. With these optimizations, our scalability increased by more than a factor of 2 as GridDB's optimized implementation kept a 128-node cluster near full-utilization.

6.2.1 Results: Baseline Implementation

First, we characterize the scalability of GridDB's baseline implementation. Figure 9 shows a node-utilization profile (node-utilization verses time) for our 16-node, 32-image run. The graph shows that GridDB is able to keep the cluster fully-utilized. Tiny kinks in the top of the graph show (predominantly) sub-second intervals after a job completes on a node, and before the next job is dispatched to it. This initial graph can be contrasted with the node-utilization profile for our 64-node, 128-image run, depicted in Figure 10. In this graph, GridDB struggles to keep the cluster 50% utilized.

To understand these scalability problems, we briefly ex-

	unfold		fold		
Program	Queries	Links (leaf-linking)	Queries	Links (root-linking)	Average Exec Time (s)
mscpipe	13	788	297	2	273
photpipe	14	2	25	1	342

Table 1: [EXP3] Characteristics of the SuperMACHO full-image pipeline

	unfold			fold		
Program	Queries	Links (leaf-linking)	Links (smart-linking)	Queries	Links (smart-linking)	Average Exec Time (s)
mscpipe1	13	788	1	45	5	75
mscpipe2	17	876	50	23	5	168
mscpipe3	17	861	70	35	5	34

Table 2: [EXP3] Characteristics of the sub-programs (per-image) of a decomposed mscpipe.

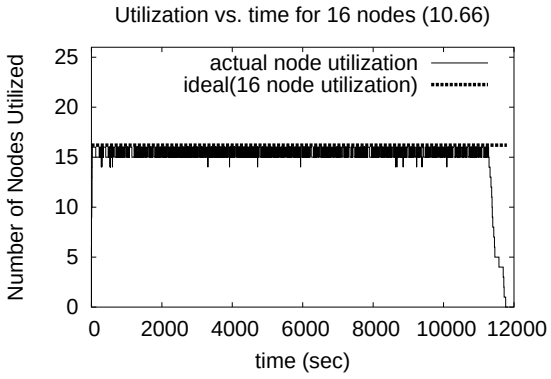

Figure 9: [EXP1] GridDB's baseline implementation is able to achieve full node-utilization on a small, 16-node cluster.

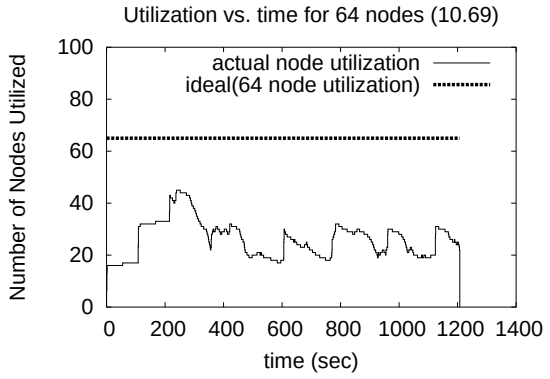

Figure 10: [EXP1] The baseline implementation cannot keep 64 nodes busy. Node utilization is often less than 50%.

plain the implications of handling many requests concurrently within GridDB. Because our cluster can run multiple processes simultaneously, it is also the case that multiple processes need to be unfolded (or folded) simultaneously. In a concurrent environment, two factors contribute the latency of a stage: contention and queuing. Contention is caused by multithreading. GridDB uses multiple threads in the unfold and fold stages to interleave requests, which help hide the latencies of blocking calls to the file system and database. As the number of concurrent requests increases, multithreading has the positive effect of increasing throughput (requests/sec) through the stage, but the negative effect of increasing the average service-time (time for a thread to complete a request) of each individual request. At a certain point thrashing sets in, and adding additional threads actually *decreases* the throughput of the system. Since our goal is to unfold and fold processes as fast as, or faster than the cluster can execute them, our goal was to maximize throughput. Towards this end, we ran experiments to choose a good "sweet-spot" for the number of threads in both the fold and unfold phases. We settled on 16 unfold threads and 8 fold threads. This was reasonable, as the unfold stages acts more frequently as a bottleneck.

The second factor, queuing, emerges when a request is ready to be processed, but no threads are available. As a result the job is unable to enter its intended stage and waits in a queue until a thread is freed and becomes available. We define the *stage time* as the sum queuing time and service time for a particular stage.

In the absence of concurrency (in a 1-node run), the unfold stage requires 33 seconds, and the fold stage requires 10 seconds. With 64-nodes, the unfold and fold stage times are much higher (Figure 11). In fact, they are higher than the execution times of the program, with the unfold stage representing the primary bottlneck (requests, on average, spend 2500 seconds in the unfold stage). We decomposed stage time into queuing time and service time, as shown in Figure 12. A large portion of time is spent queuing. We should note that queuing time is not necessarily a sign that the cluster is under-utilized. It is possible that the cluster is fully-utilized, and that offered load is just very high. In this particular scenario, the cluster is *not* fully-utilized. Our strategy for reducing stage time was simply to reduce the service time for the unfold stage. By reducing service time, we also indirectly reduce queuing time. By spending less time per request, each thread can process requests at a higher rate, moving requests through the queue faster.

6.2.2 Profiling Drill Down

As mentioned, the unfolding of mscpipe processes was not occurring at a fast enough rate to keep the 64-node cluster utilized. To mitigate this deficiency, we drilled down on a profile of the mscpipe unfold stage. The results are shown in Figure 13, where file links account for 64% of the ser-

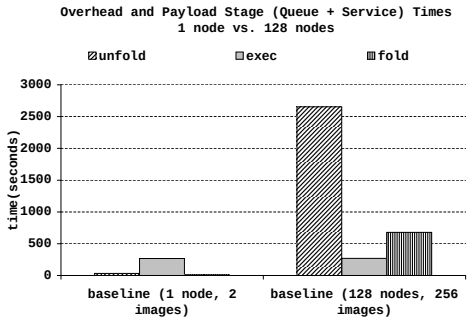

Figure 11: [EXP1] Stage times in overhead stages (unfold and fold) and the exec stage. Comparison of overheads in a 1-node run vs. a 64-node run.

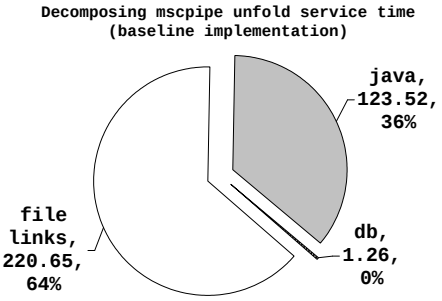

Figure 13: [EXP1] A profile of how time is spent, on average, in the unfold phase of mscpipe (128-nodes).

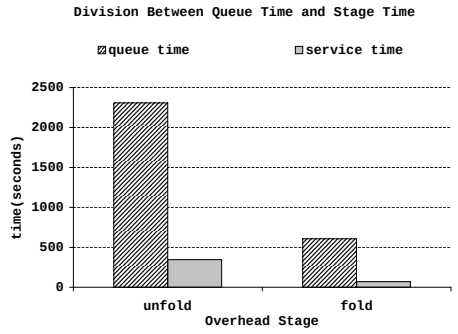

Figure 12: [EXP1] Time spent in queue of overhead stage vs. being serviced by overhead stage (64-nodes).

vice time. We were rather surprised, in fact, that database processing queries amounted to less than 0.5% of overall processing. The low cost of database access to the fact that:

1. PostgreSQL and its associated database driver was quite fast (queries are sub-millisecond and transactions are less than 30 milliseconds).
2. Linking is needed at a high frequency, and is relatively slow on Lustre (see Section 6.5).

6.2.3 *Linking Optimizations*

In light of the linking bottleneck, we applied a pair of optimizations. Overall linking time can be decomposed into two factors, the number of links being created and the time consumed per link. Each of our optimizations reduce one of these factors.

Reduce Time/Link with Batching: Recall from Section 5, our implementation relies upon the `ln` utility to create links. Instead of calling the `ln` each time a link needed to be created, we batched all of the linking requests. We stored a list of links that needed to be created and executed a process that creates all the links (via system calls) at once. Benchmarks showed that each `ln` call costs about 35 ms, and that the batch program creates links at a cost of (120 + 14) ms per link. Even with this bias, the batched program performs better than the non-batched program.

Reduce Number of Links with Smart-Linking: Section 4.1 explained why leaf-linking is required in order to preserve filesets. Leaf-linking is never necessary during a fold phase, but possibly necessary during a process' unfold phase. We identified the filesets where leaf-linking was not necessary (when the process does not add to a fileset) and transmitted them through root-linking. This is a hybrid scheme, which we call smart-linking. Figure 14 illustrates the smart-linking protocol. Directory A is not modified by program G, so it is root-linked. Program G adds a file to directory B, so it is leaf-linked.

Smart-linking is especially effective in the FullPipe workflow. Many links are created to reference the files in the Reference Images fileset. The directory represented by the Reference Images fileset is not modified, and so can be root-linked. Our profiling indicates that the number of links in the unfold stage of the mscpipe program drops from 788 to 2.

We profiled these two optimizations on a 1-node, 1-image run. Their effectiveness is shown in Figure 15. Either optimization results in substantial savings while the combination of both optimizations yields additional time savings.

6.3 Results: With Optimizations

After applying the linking optimizations, we ran scale-up experiments and saw a significant improvement. GridDB, with optimizations, is able to scale-up more than twice as far as GridDB, without linking optimizations. The optimized version capable of keeping 128 nodes fully occupied, as shown in Figure 16. Drilling down on the run's profile, the major cause was a large reduction in the unfold stage time of mscpipe. A comparison between stage times of the baseline and optimized implementations is shown in Figure 17. Although the optimized GridDB run is managing twice as many nodes and processing twice as many images, its unfold stage time is 10 times less than the baseline version of GridDB leading to increased scale-ups.

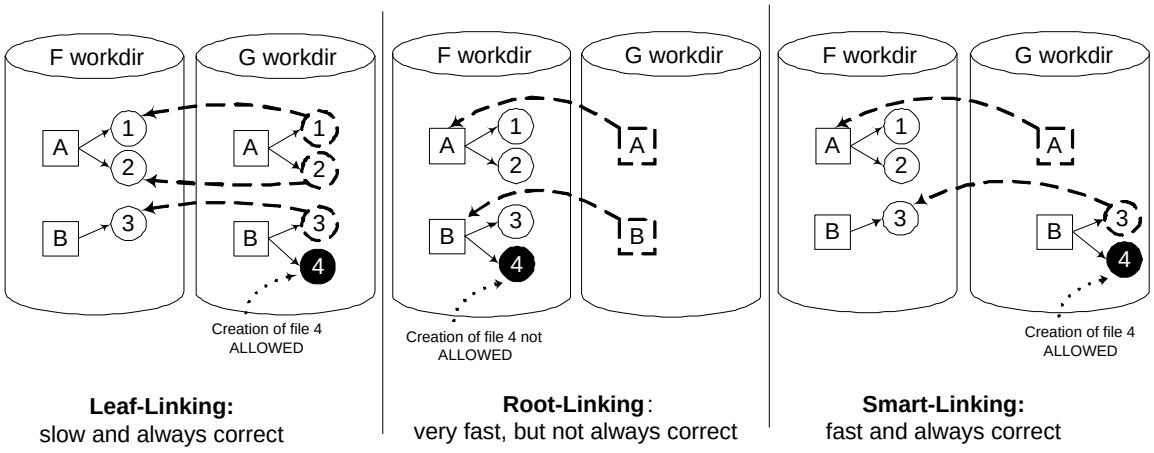

Figure 14: The smart-linking file transfer discipline is uses root-linking when possible and leaf-linking when necessary.

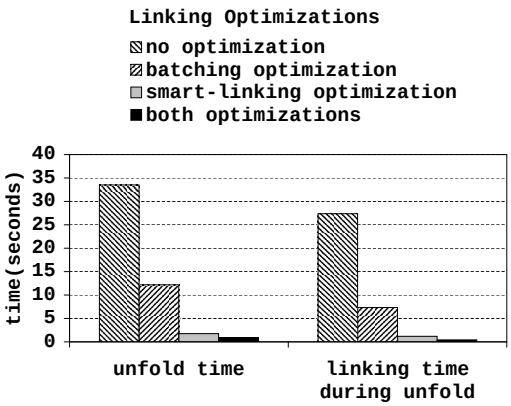

Figure 15: [EXP1] Batching and smart-linking both improve unfold service. Their combination does even better. These profiles were taken on a (1 node, 1 image) run.

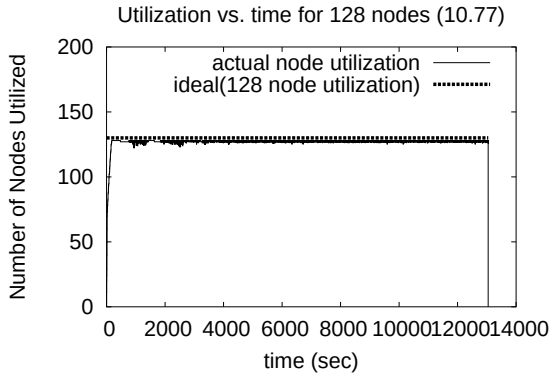

Figure 16: [EXP1] Utilization plot after optimizations. GridDB is able to keep 128 nodes busy.

6.4 Exp 2: Varying Middleware-Load

Our second set of experiments are aimed at demonstrating the sensitivity of middleware performance with respect to f workflow program characteristics. In these experiments, wee decomposed a portion of our SuperMACHO workflow into a fine-grained 3-node workflow and executed it on our optimized GridDB implementation. The 3 programs have a high middleware-load, imposing a larger number queries and file system links, and executing in a shorter amount of time. In these experiments, we demonstrate that even with our optimized implementation, GridDB's scale-up ability is sensitive to the middleware-load of the workflow programs it is executing.

Since this workflow is made of programs with a higher middleware-load than FullPipe, GridDB is relatively strained in executing it. Figure 19 shows the utilization profile for the execution of 256 images on a 128-node cluster through DecomposedPipe. The execution is carried in batch: all mscpipe1 processes are executed first, followed by mscpipe2, and then mscpipe3 processes. The execution can be divided into three distinct stages characterized by the program being executed and delineated by time=900 and time=1600. As the programs increase in middleware-load, GridDB has more trouble keeping the cluster occupied.

This test demonstrates that, unfortunately, GridDB (and more generally, other middlewares that intervene in workflow processing) is sensitive to the middleware-load of their programs. In Section 7, we provide guidelines for how users may best use their middleware to minimize middleware-load and maximize middleware benefit.

6.5 Exp 3: Varying File Systems

In our last set of experiments, we compared GridDB's performance when executing over the two network file systems available on MCR, Lustre and MCR. Lustre [8], is an open-source parallel network file system for linux clusters. NFS [17] is a widely-used, but centralized, network file system.

Our test are based on a 5-stage pipeline similar to Decom-

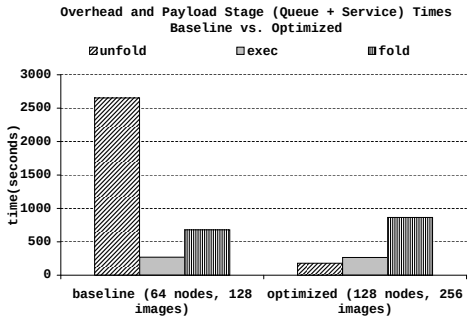

Figure 17: [EXP1] Comparison of stage times in baseline vs. optimized implementations. Unfold stage time is down 10-fold in the optimized implementation.

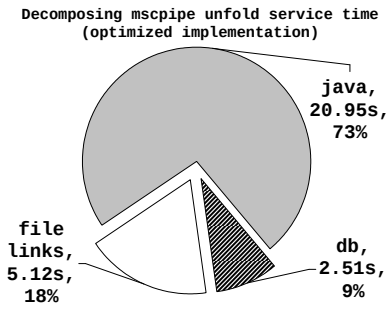

Figure 18: [EXP1]: The unfold profile for our (128 nodes, 256 images) run shows a marked improvement in file-linking, and also an improvement in absolute java execution time (Compare with Figure 13).

posedPipe with stages that needed on average, 75 links and 25 queries in the unfold stage, and 25 queries and 5 links in the fold phase. The stages, on average, required 80 seconds to execute (on Lustre).

In our tests, Lustre out-performed NFS on the Exec phase (science code execution), as shown in Table 3. This was expected, as Lustre parallelizes reads and writes. We were, however, surprised that Lustre was significantly slower on in the fold and unfold phases. The deficit was traced back to the metadata operations symlink(2) and stat(2) operations which are used by GridDB for file linking operations. These operations are much faster on NFS (see table 4.

According to LLNL systems experts, Lustre's highly distributed architecture, strict cache coherency constraints, and limitations of this particular version of the Lustre code conspire to make file system metadata operations particularly slow. NFS service on the other hand is not distributed and implements weak cache coherency, thus can turn around metadata requests quicker.

Application users will prefer to use Lustre where it is available, although GridDB will execute more scalably on NFS. The middleware-load of programs running on NFS is reduced by two factors: programs run longer and fold and unfold operations cost less. In some situations, where NFS will be the only choice (e.g. home-grown clusters), we expect that GridDB will scale better than the results we have reported in Section 6.2.

6.6 Summary of Experiments

In these experiments, we show that our baseline implementation of a single-node GridDB encounters scalability problems while managing a 64-node cluster. Through a series of bottleneck drill-downs, we identify that 70% of overheads occur during calls from GridDB to the file system. We propose a pair of optimizations, batching and "smart-linking," to drastically reduce interactions between our middleware and the file system. As a result, GridDB is able to manage 128-nodes, with almost 100% utilization. In a second set of experiments, we demonstrate the importance of a program's middleware-load, or the load that is induced onto GridDB by a workflow program. Because modeling is an important factor in middleware (and science application) performance, we provide guidelines for making a decision on the modeling of workflows. In our final set of experiments, we show the sensitivity of GridDB's performance with respect to the underlying file system. While a parallel file system such as Lustre exhibits better application performance, its metadata operations are slower than a centralized network file system, such as NFS. Both factors, faster application execution, and slower linking operation make GridDB less scalable by virtue of increasing the "middleware-load" of workflow programs. We expect that in situations when NFS is that *only* file system available, GridDB should perform and scale better.

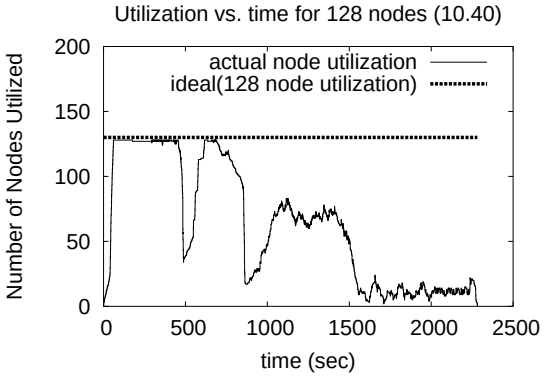

Figure 19: [EXP2] Sensitivity to Middleware-load.

7. MODELING IMPLICATIONS

In Section 6.4, we established a connection between GridDB's scalability and a program's middleware-load. In this section, we discuss how a modeler can control middleware-load, making precision trade-offs between the use and omission of middleware processing.

One method of reducing middleware load is to model program interfaces more precisely, in contrast to the "lightweight modeling" discussed in Section 4.2. For example, if a pro-

Quantity	NFS(s)	Lustre(s)	NFS/Lustre
Total Exec Time	2235	412	5.42
Total Unfold Time	24	65	0.37
Total Fold Time	7	27	0.26

Table 3: Comparison of stages when run in NFS and Lustre. Run on a 5-stage pipeline similar to DecomposedPipe.

File System	stat	symlink
NFS	< 1 ms	< 1 ms
Lustre	5.2 ms	3.3 ms

Table 4: Comparison of metadata operations used by GridDB on NFS and Lustre using microbenchmarks.

cess creates files `out/1`, `out/2` and `out/3`, and only needs to export `out/1`, then the modeler should specify the precise file to export. As discussed in Section 4.2, this requires additional understanding of the program from the modeler and reduces the maintainability of the workflow model.

A second method is the use of a coarse grain model, encapsulating more execution in one program, lengthening its runtime and eliminating overhead stages. In the next section, we discuss the trade-offs between coarse- and fine-grain models.

7.1 Coarse- vs. Fine-Grain Modeling

In this section, we consider the decision between coarse- and fine-grain modeling. one which decouples two programs, interjecting their executions with middleware processing, and the other, tightly coupling two programs.

Figure 20 illustrates the two alternatives. In the coarse grain model, one program, which executes both programs F and G (e.g. a script calling the programs in sequence), is provided to GridDB. GridDB is unaware of any separation between programs F and G, and does not interject between their processing. To the middleware, the effect of applying program $FAndG$ to fileset X is the creation of fileset Z.

In the first scenario, a description of both programs F and G have been provided to the middleware. The pipeline is the composition of the two programs. In this model, the

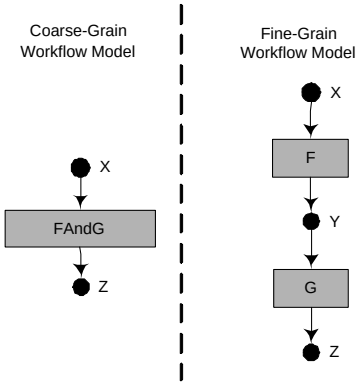

Figure 20: Two Modeling alternatives for a simple 2-node workflow

effect of applying program F to input X creates data fileset Y. The application of G to fileset Y creates fileset G.

In the coarse-grain model, there is one less unfold and fold overhead stage (between F and G). This lessens the middleware-load of the entire workflow. There are a set of benefits, however, that are lost in the coarse-grain model, which we discuss these next.

7.2 Benefits of Fine-Granularity Modeling

The benefits of the fine-grain model may be grouped into two categories, those associated with the *decoupling* of F and G and those stemming from middleware *interjection* between F and G.

7.2.1 Decoupling

A handful of activities concerning either program F or program G may be decoupled. The activities include: execution, prioritization, caching, resource estimation and allocation.

Execution, Prioritization and Caching: the execution of programs F and G are separated. This may be beneficial when one wants the benefits of one program without paying the costs of another. For example, a user may want to produce Y without executing G. This is particularly useful if G is long-running. Likewise, program F and G can be cached, or prioritized independently.

Resource estimation and allocation: The resource estimation of programs F and G can be made separately, and their execution can be assigned to different resources when beneficial. If programs F and G have vastly different requirements, it may be easier to assign their executions independently rather than finding one compute-node that satisfies the requirements of both.

7.2.2 Interjection

A second set of benefits involves GridDB's *interjection* between the two programs. Examples here include monitoring and steering of computations.

Monitoring: Monitoring can occur between F and G. For example, if the execution of F falls outside of three standard deviations of the expected execution time, the middleware may issue a warning to the user, or automatically 'retry' the computation on another node. The middleware may also 'ingest' results from F into a database where a user may begin querying before G completes.

Steering: the user (or GridDB) may 'steer' the pipeline computation based on the intermediate result Y. For example, if a particular Y indicates that the pipeline execution will be interesting, the user middleware may expedite the processing of Y.

8. RELATED WORK

Recently, there have been a handful of systems similar to GridDB, which elevate data to a first-class status within the context of workflow management [20, 9, 7]. Most of the work, thus far, has focused on defining languages, interfaces and features.

The Pegasus project has begun addressing issues of workflow middleware performance. In [11], they identify that middleware can substantially increase the overhead of workflow execution, if not used properly. In that paper, they adjust parameters of the condor batch scheduler and show that overall execution time of a fine-grain workflow is highly sensitive to how often particular activities, such as scheduling and job dispatch, are performed. In [10], examines the trade-offs between using the Pegasus middleware system and

MPI. The former is easier to program with while the latter should yield better performance. They show that because of features such as computation caching, the Pegasus implementation is able to at least match the MPI implementation. Our work differs from these in that we address performance and scalability in the context of higher-order functions such as data provenance, computation caching, and scheduling.

9. CONCLUSION

In this work, we demonstrate the efficacy of GridDB, a data-centric scientific workflow system, in scalably executing a data-intensive image processing application on a cluster computer. Contrary to our original assumptions, middleware calls against the file system, rather than the database, constituted initial bottlenecks in scaling the middleware. We implemented techniques to circumvent these interactions, more than doubling GridDB's scalability. Secondly, while software may be written in a scalable manner, application performance is sensitive to the middleware-load of workflow programs. We provide guidelines for middleware users to trade-off costs for specific middleware benefits. Lastly, we observed the performance of GridDB on the NFS and Lustre file systems. Because workflow programs will exhibit lower middleware-load when executed over NFS, the scalability of GridDB when executing on NFS will be better than when executing on Lustre.

More work needs to be done. Profiling our optimized system indicates that the java middleware code is the emergent bottleneck. We plan to optimize this code using standard java optimization techniques. Then, we would like to explore the parallelization of the GridDB server across multiple nodes on the cluster computer.

10. ACKNOWLEDGEMENTS

We thank Marcus Miller for discussions and the initial port of GridDB onto MCR, Sergei Nikolaev for assistance with the SuperMACHO image processing codes and Brian Behlendorf for enlightening us with Lustre file system internals.

11. REFERENCES

[1] Apache Derby Home Page. `http://incubator.apache.org/derby/`. Accessed 4/24/05.

[2] hsqldb Home Page. `http://hsqldb.sourceforge.net/`. Accessed 4/24/05.

[3] MySQL Home Page. `http://dev.mysql.com/`. Accessed 4/24/05.

[4] PostgeSQL Home Page. `http://www.postgresql.org/`. Accessed 4/24/05.

[5] The large synoptic survey telescop (lsst), 2003.

[6] Supermacho home page. `ttp://www.ctio.noao.edu/~supermacho/`, 2005.

[7] C.A. Goble, S. Pettifer, R. Stevens and C. Greenhalgh. *The Grid: Blueprint for a New Computing Infrastructure Second Edition*. Morgan Kaufman, 2003.

[8] Cluster File Systems, Inc. Lustre: A Scalable, High-Performance File System. Technical report, 2002. `http://www.lustre.org/docs/whitepaper.pdf`.

[9] E. Deelman, *et al.*. Pegasus : Mapping scientific workflows onto the grid. In *Across Grids Conference 2004*. 2004.

[10] Deelman Et Al. A Comparison of Two Methods for Building Astronomical Image Mosaics on a Grid. In *The 7th Workshop on High Performance Scientific and Engineering Computing*. 2005.

[11] G. Singh AND C. Kesselman AND E. Deelman. Optimizing Grid-Based Workflow Execution.

[12] Grid physics network (griphyn) white paper, 2003.

[13] R. Izmailov, *et al.*. Fast parallel file replication in data grid. In *Workshop proceedings, The Future of Grid Data Environments*. 2004.

[14] Jim Gray. Distributed Computing Economics. `ftp://ftp.research.microsoft.com/pub/tr/tr-2003-24.pdf`. Accessed 04/22/05.

[15] D. T. Liu *et al.*. The design of griddb: A data-centric overlay for the scientific grid. In *VLDB*, pp. 600–611. 2004.

[16] M. Livny, *et al.*. Particle physics data grid collaboratory pilot. `http://www.ppdg.net/docs/SciDAC/PPDG_overview.pdf`, September 2001.

[17] B. Pawlowski, *et al.*. NFS version 3: Design and implementation. In *USENIX Summer*, pp. 137–152. 1994. URL `citeseer.csail.mit.edu/pawlowski94nfs.html`.

[18] Susan L. Graham AND Mark Snir. The NRC Report on the Future of Supercomputing. In *Cyber Technology Watch Quarterly*. 2005.

[19] Yolanda Gil AND Ewa Deelman AND Jim Blythe AND Carl Kesselman AND Hongsuda Tangmunarunkit. Artificial intelligence and grids: Workflow planning and beyond. In *In IEEE Intelligent Systems: special issue on e-science*. 2004.

[20] Y. Zhao, *et al.*. Chimera: A virtual data system for representing, querying, and automating data derivation. In *14th Conference on Scientific and Statistical Data Management*. 2002.

[21] Y. Zhao, *et al.*. e Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR*. 2002.

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under contract W-7405-Eng-48.